# Leveraging Datapath Propagation in IC3 for Hardware Model Checking

Hongyu Fan, and Fei He, *Member, IEEE*

***Abstract*—IC3 is a famous bit-level framework for safety verification. By incorporating datapath abstraction, a notable enhancement in the efficiency of hardware verification can be achieved. However, datapath abstraction entails a coarse level of abstraction where all datapath operations are approximated as uninterpreted functions. This level of abstraction, albeit useful, can lead to an increased computational burden during the verification process as it necessitates extensive exploration of redundant abstract state space.**

**In this paper, we introduce a novel approach called datapath propagation. Our method involves leveraging concrete constant values to iteratively compute the outcomes of relevant datapath operations and their associated uninterpreted functions. Meanwhile, we generate potentially useful datapath propagation lemmas in abstract state space and tighten the datapath abstraction. With this technique, the abstract state space can be reduced, and the verification efficiency is significantly improved. We implemented the proposed approach and conducted extensive experiments. The results show promising improvements of our approach compared to the state-of-the-art verifiers.**

***Index Terms*—datapath abstraction, datapath propagation, hardware verification, reachability safety**

## I. Introduction

IC3 algorithm [1] (also known as PDR [2]) is one of the most successful and widely applied technique for verifying reachability of safety property in hardware model checking. The IC3 algorithm incrementally explores the design's state space and tries to construct a proof of correctness. However, bit-level IC3 algorithms [3]–[5] suffer from the state space explosion problem. As the bit-width and complexity of the hardware design increase, the IC3 algorithm's performance degenerates rapidly, threatening its scalability.

In Verilog RTL design, the datapath is responsible for processing and manipulating data as it flows through the system. Datapath operations refer to the specific procedures performed within the datapath, typically involving arithmetic, logical, and data movement operations. They are the most essential units in the design. Complex functions usually consist of many datapath operations. A popular approach [6]–[8] integrates the IC3 algorithm with datapath abstraction [9], which denotes datapath operations as uninterpreted functions (UF) instead of their explicit implementation details. Therefore, UFs serve as over-approximations for datapath operations. Importantly, the verification process does not entail defining the specific logic or functionality encapsulated within these functions. This abstraction facilitates a higher level of generality, enabling a more abstract representation of the overall datapath behavior.

Constraint solving for UF [10] is much faster than bit-vector (BV) [11], [12]. Therefore, each IC3 call on abstract state space is mostly several orders of magnitude more efficient than the bit-level IC3 call. However, abstraction may bring spurious counterexamples. For each abstract counterexample returned by IC3, the *counterexample-guided abstraction refinement* (CEGAR) [13] procedure checks if the abstract counterexample is spurious. If it is, then the refinement procedure generates *datapath refinement lemmas* to prune the abstract state space and tighten the current abstraction. Each datapath lemma is a constraint formula over UFs. Then, the verification procedure calls the IC3 iteratively until either the property holds or a real counterexample is found.

Although integrating the IC3 with datapath abstraction and refinement is shown to be successful and practical [14], [15], the iteration of the CEGAR is also a crucial factor to the overall efficiency of the verification process. However, the CEGAR is designed as a general framework for abstraction-based verification. The knowledge of datapath operations cannot be fully utilized. Roughly abstracting all the datapath operations as uninterpreted functions makes the verification procedure lose all the semantics of datapath operations. Constraint solving can assign arbitrary values to UFs. Therefore, the verification procedure may find numerous spurious counterexamples, even if they are trivial, and require many rounds of refinement to tighten the abstraction. On the other hand, applying this knowledge in CEGAR may be useful for pruning the abstract state space and thus improving the overall efficiency.

Our basic idea is to utilize the knowledge of datapath operations in abstract state space by propagating the accurate values to the corresponding UFs and guiding the datapath abstraction-based IC3 for hardware verification. A straightforward attempt for this idea is constant propagation [16], [17], an optimization technique that aims to identify and propagate constant values throughout a system. It replaces variables or expressions with their known constant values, eliminating unnecessary computations and improving runtime performance. However, constant propagation is performed in the concrete state space.

In this paper, we propose a *datapath propagation* that propagates constant values from concrete state space to abstract state space. In more detail, for each abstract constraint formula, we first recognize constant values and related UFs. Then, we consider the original semantics of UFs, i.e., their corresponding datapath operations. We propagate constant values to drive the outcomes of these datapath operations and

Fei He is the corresponding author. The authors are with the School of Software, KLISS, BNRist, Tsinghua University, Beijing 100084, China. Email: fhy18@mails.tsinghua.edu.cn, hefei@tsinghua.edu.cn.

This work was supported in part by the National Natural Science Foundation of China (No. 62072267 and No. 62021002).

assign the outcomes to the corresponding UFs. Subsequently, we substitute these UFs with their outcomes and continue the iterative propagation. With this technique, some UFs are assigned with accurate values, or relations between UFs and constants can be determined. Therefore, we prune redundant abstract space and tighten the datapath abstraction.

Moreover, we propose to generate *datapath propagation lemma* (DPL), which is another type of datapath lemma generated during the propagation process. It can be generated in two situations. First, once the propagation deduces that the current formula is unsatisfiable, we generate DPL to record the core reason and block the possible spurious counterexample. Second, if some predicates or binary relations referring to datapath operations are determined during the propagation, we generate DPL to record this information and facilitate further verification. The generated DPLs can also eliminate spurious counterexamples and, more importantly, reduce the number of CEGAR iterations. Our method is performed in the abstract state space, independent of CEGAR. Combining our approach and refinement in CEGAR can further prune the abstract state space. The verification efficiency is thus improved.

We implemented the proposed method on top of AVR [14], which is the champion tool of the latest Hardware Model Checking Competition (HWMCC). Our implementation is called AVR$_{\text{dp}}$. We collected benchmarks from HWMCC 2019 and 2020 – the last two competitions. We compare AVR$_{\text{dp}}$ with state-of-the-art hardware verification tools, including AVR, IC3IA [18], Pono [19], ABC [3], nuXmv-ic3 [20], and Avy [4]. The experimental results show that AVR$_{\text{dp}}$ solves more cases than competitor tools. Counting on both-verified cases, AVR$_{\text{dp}}$ achieves 1.46x, 20.04x, 2.01x, 2.09x, 2.84x, 3.65x speedup over AVR, IC3IA, Pono, ABC, nuXmv-ic3, and Avy, respectively. Especially, AVR$_{\text{dp}}$ generates 3923 datapath propagation lemmas and reduces 29.5% refinements than AVR.

The contributions of this paper are summarized as follows:

- We proposed a novel datapath propagation approach in abstraction-based IC3 for hardware verification.
- We devised a datapath propagation lemma generation procedure, with which the deduced results can be kept to facilitate further verification.
- We implemented the proposed method on top of AVR and conducted extensive experiments. Experimental results show the promising performance of our approach.

The rest of this paper is organized as follows. Section II introduces necessary preliminaries. Section III uses an example to motivate our approach. Section IV details the datapath propagation. Section V shows how to combine datapath propagation with IC3. Experimental results and analysis are presented in Section VI, followed by related works in Section VII and conclusion in Section VIII.

## II. Preliminaries

### A. Notations

In *first-order logic* (FOL), a *term* is a variable, a constant, or an $n$-ary function applied to $n$ terms; an *atom* is $\bot$, $\top$, or an $n$-ary predicate applied to $n$ terms; a *literal* is an *atom* or its negation. A *cube* is a conjunction of literals and a *clause* is a disjunction of literals. A *first-order formula* is built from literals using Boolean connectives and quantifiers. An interpretation (or model) $M$ consists of a non-empty object set $dom(M)$, called the *domain* of $M$, an assignment that maps each variable to an object in $dom(M)$, and an interpretation for each constant, function, and predicate, respectively. A formula $\Phi$ is *satisfiable* if there exists a model $M$ so that $M \models \Phi$; $\Phi$ is *valid* if for any model $M$, $M \models \Phi$.

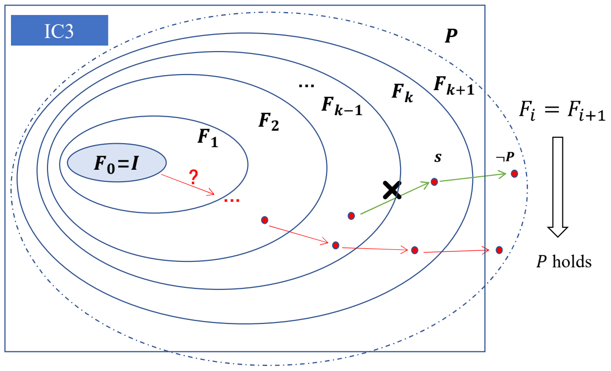

Fig. 1. Overview of the IC3 algorithm.

A *first-order theory* $\mathcal{T}$ is defined by a *signature* and a set of *axioms*. The *signature* consists of constant symbols, function symbols, and predicate symbols allowed in $\mathcal{T}$; the *axioms* prescribe the intended meanings of these symbols. A $\mathcal{T}$*-model* is a model that satisfies all axioms of $\mathcal{T}$. A formula $\Phi$ is $\mathcal{T}$*-satisfiable* if there exists a $\mathcal{T}$-model $M$ so that $M \models \Phi$; $\Phi$ is $\mathcal{T}$*-valid* if it is satisfied by all $\mathcal{T}$-models. The *satisfiability modulo theories* (SMT) [10], [21] problem is a decision problem for formula $\phi$ in some combination of first-order theories. For each theory $\mathcal{T}$ in $\phi$, there is a $\mathcal{T}$-solver that can check the $\mathcal{T}$-satisfiability of conjunctions of literals in $\mathcal{T}$.

### B. Model Checking

A hardware design can be represented as a netlist or a model in a hardware description language such as Verilog. Let $X$ be the set of state variables in the design; let $X'$ be the primed copy of $X$ representing the next-state variables. The design's behavior can be encoded as a model checking problem via a 4-tuple $\mathcal{P} := \langle X, I, T, P\rangle$, where $I(X)$ is a formula for the initial states, $T(X, X')$ is a formula for the transition relation, and $P(X)$ is a formula for the desired safety property. Specifically, the next-state variables in $T$ are represented as functions of present-state variables. Input variables are treated as state variables whose next states are unconstrained.

A *state* $s$ is an assignment to all variables in $X$. A *trace* is a sequence of states $s_0, s_1, \ldots, s_k$ such that $I(s_0)$ holds, and $T(s_i, s_{i+1})$ holds for $0 \le i \le k-1$. The property formula $P$ asserts that all reachable states satisfy $P$, i.e., $P$ should be invariant for the design. Otherwise, there must be a finite trace $s_0, s_1, \ldots, s_k$, which is a counterexample that $P(s_k)$ does not hold. An *inductive invariant* $F$ is a formula satisfying: (1) $I \to F$, and (2) $F$ is closed under the transition relation, i.e., $F \wedge T \to F'$.

**Algorithm 1** IC3 $(I, T, P)$

```
1:  if $I \wedge \neg P$ or $I \wedge T \wedge \neg P'$ then
2:      return counterexample trace;
3:  $k = 1, F_k = P$;
4:  while true do
5:      $F_{k+1} = P$;
6:      while $F_k \wedge T \wedge \neg P'$ do
7:          let $s$ be the satisfying assignment;
8:          if Reachable$(s, I)$ then
9:              return counterexample trace;
10:         else
11:             Block$(s, k+1)$;
12:     if $F_i = F_{i-1}$ for some $2 \leq i \leq k+1$ then
13:         return empty trace; // $P$ hold
14:     $k$++;
```

### C. IC3 Algorithm

IC3 (or PDR) is a well-known algorithm for determining whether a hardware design satisfies a given safety property $P$. It represents a major advance over previous SAT-based induction methods [22]–[24]. Alg. 1 lists the pseudocode of IC3. Taking $I, T, P$ as input, it first looks for 0-step and 1-step counterexample traces (line 1). If none are found, the algorithm instantiates frontier $F_k$, the over-approximation of $k$-step reachable states ($k \geq 1$), to $P$ (line 3). Then, the IC3 algorithm iteratively checks if $F_k$ can reach $\neg P$ states in one transition (line 6). Each satisfying assignment $s$ is checked to determine if it is reachable from $I$ (line 7). If unreachable, e.g., the green trace in Fig. 1, $s$ is blocked and used to tighten the frontiers $F_1$ to $F_{k+1}$ (line 10). This check continues until either a counterexample trace of the length $k+1$ is found (line 8), or all the states that violate $P$ in one transition are unreachable from $I$. If the algorithm finds $F_i = F_{i-1}$ for some $2 \leq i \leq k+1$, it returns an empty trace indicating that $P$ holds (line 11), and $F_i$ is an inductive invariant that satisfies $P$ ($F_k \rightarrow P$). Otherwise, the IC3 algorithm increments $k$ and continues to check the existence of counterexample traces on longer transitions (line 13).

The pseudocode shows a sketch of the algorithm and hides many details. It consists of numerous 1-step backward reachability checks processed in order. These reachability checks are represented as formulas using the BV theory. Therefore, the state space is exponential in the bit width of state variables. Given $\mathcal{P}$ with $n$ total bits, each bit can be 0 or 1, and there are up to $2^n$ states in the concrete state space. Therefore, as the bit width increases, the scale of the state space grows exponentially, and the efficiency of IC3 degenerates rapidly.

### D. Datapath Abstraction

Abstraction is a common technique for improving the efficiency and scalability of verification. It creates an abstract model that captures the critical behavior and properties of the system while approximating certain details as needed. The abstract model brings higher-level representation and simplified views of the original system. In this way, an abstract state can represent a cluster of concrete states. Therefore, it reduces the proof of a property on an infinite or large concrete state space to a proof on an abstract state space.

```verilog
module Example (clk);
   input wire clk;
   reg [1:0] x, y;
   initial begin
      x = 2'd0;  y= 2'd0;
   end
   always @ (posedge clk) begin
      x <= (x<y)? x : (y!=x)? y : x+1;
      y <= (y==x)? y+1 : (x<y)? y : x;
   end
   wire P = (y<=x);
endmodule
```

Fig. 2. Verilog description of an example sequential circuit with a specified safety property. The state variables are 2-bit unsigned integers $x = x_1x_0$ and $y = y_1y_0$. The main sequential logic involves computing the next-state values of $x$ and $y$. The safety property asserts that $y \leq x$ is always satisfied.

Datapath abstraction [6] replaces state variables and datapath operations with uninterpreted functions (UF). It returns the abstract version of the original problem $\mathcal{P} := \langle X, I, T, P\rangle$ as $\hat{\mathcal{P}} := \langle \hat{X}, \hat{I}, \hat{T}, \hat{P}\rangle$. $\hat{\mathcal{P}}$ over-approximates the original system and is a sound abstraction, i.e., if $\hat{P}$ is proved safe on the abstract state space, so is $P$ on the concrete state space. However, a counterexample that violates $\hat{P}$ on the abstract state space may be spurious on the concrete state space due to the coarse abstraction.

## III. Motivation

This section motivates our approach with a simple example. We introduce IC3 with datapath abstraction and emphasize the overlooked significance of datapath operation knowledge.

### A. IC3 with Datapath Abstraction

Consider the example design in Fig. 2. The design's behavior can be encoded as a model checking problem $\mathcal{P}$ where $x, y$ are state variables and $I, T, P$ are:

$$\begin{aligned}
&I : x = 0 \;\wedge\; y = 0\\
&T : x' = (x < y) \;?\; x : (y \neq x) \;?\; y : x + 1 \;\wedge\\
&\qquad y' = (y == x) \;?\; y + 1 : (x < y) \;?\; y : x\\
&P : y \leq x
\end{aligned}$$

IC3 can be enhanced with the datapath abstraction. The enhanced algorithm, called DP-IC3 [6], is shown in Alg. 2. It first calls DP-Abstract to perform the datapath abstraction (line 1). For the example design, we have:

$$\begin{aligned}
&\hat{I} : \hat{x} = \hat{0} \;\wedge\; \hat{y} = \hat{0}\\
&\hat{T} : \hat{x}' = LT(\hat{x}, \hat{y}) \;?\; \hat{x} : (\hat{y} \neq \hat{x}) \;?\; \hat{y} : ADD(\hat{x}, \hat{1}) \;\wedge\\
&\qquad \hat{y}' = (\hat{y} == \hat{x}) \;? ADD(\hat{y}, \hat{1}) : LT(\hat{x}, \hat{y}) \;?\; \hat{y} : \hat{x}\\
&\hat{P} : LE(\hat{y}, \hat{x})
\end{aligned}$$

$\hat{\mathcal{P}}$ uses uninterpreted sort and converts datapath operations (e.g., $<, +$) with UFs (e.g., LT, ADD). Note that state variables or constants are denoted as 0-ary UFs.

Then, IC3 runs on the abstract state space. Let $\Phi_{drl}$ be the conjunction of datapath refinement lemmas derived in CEGAR. It is initialized to $true$ (line 2). Lines 3-11 are the main body of DP-IC3. It calls the modified version of IC3 in

Alg. 1 that operates on abstract formulas. Note that $\Phi_{drl}$ serves as the fourth argument and augments all the queries that IC3 performs. If IC3 returns an empty trace, no counterexample is found in the abstract or concrete state space. Alg. 2 terminates with the conclusion that $P$ holds (lines 5-6). Otherwise, a non-empty trace representing an abstract counterexample (ACEX) is found. DP-IC3 calls DP-Concrete to generate CEX as the bit-level version of ACEX and checks its feasibility (line 8). If CEX is feasible, Alg. 2 returns CEX as a counterexample trace that witnesses the violations of $P$ (line 10). If CEX is infeasible, Alg. 2 calls DP-Refine to eliminate the spurious counterexample by generating datapath refinement lemmas (line 11). Then, DP-IC3 invokes the next round of IC3.

**Algorithm 2** DP-IC3 $(I, T, P)$

```
1:  $\hat{I}, \hat{T}, \hat{P}$ = DP-Abstract($I, T, P$);
2:  $\Phi_{drl} = true$; // initialize datapath lemmas
3:  while true do
4:      ACEX = IC3($\hat{I}, \hat{T}, \hat{P}, \Phi_{drl}$);
5:      if ACEX is empty then
6:          return empty trace; // $P$ holds
7:      else
8:          CEX = DP-Concrete(ACEX);
9:          if CEX is feasible then
10:             return CEX; // $P$ fails
11:     $\Phi_{drl} = \Phi_{drl} \wedge$ DP-Refine(ACEX);
```

Consider the example design in Fig. 2, IC3 first checks 0-step safety by calling an SMT solver. $\hat{I}$ is safe iff the query formula $\hat{I} \wedge \neg\hat{P}$ is unsatisfiable. A 0-step abstract counterexample $acex_1$: $\{\hat{x} \mapsto \hat{0},\ \hat{y} \mapsto \hat{0},\ \mathit{LE}(\hat{y}, \hat{x}) \mapsto \mathit{false}\}$ is returned, where $a \mapsto b$ means $b$ is the assignment of $a$ by the SMT solver. To check its feasibility, DP-concrete returns $cex_1$ as its bit-level counterpart: $x = 0 \wedge y = 0 \wedge \neg(y \leq x)$. Then, $cex_1$ is bit-blasted and feasibility checking is performed using the BV theory solver. Apparently, $cex_1$ is BV-unsatisfiable. DP-IC3 realizes that $acex_1$ is spurious and invokes DP-Refine to eliminate it. DP-Refine can refute the spurious counterexample by adding the negation of the abstract counterexample, i.e., $\neg acex_1$ to $\Phi_{drl}$. This procedure can tighten the current abstraction and prevents $acex_1$ from appearing again.

To derive more powerful datapath refinement lemmas, DP-Refine can extract *minimal unsatisfiable subsets* (MUSes) from the concretized cube $cex_1$. An MUS is a subset of a CNF formula such that (1) a conjunction of the MUS is still unsatisfiable; (2) removing any clause from the MUS makes it satisfiable. A possible MUS of $cex_1$ is $\neg(y \leq x) \wedge y = 0$, which can be simplified to $\neg(0 \leq x)$. Since $cex_1 \rightarrow \neg(0 \leq x)$, then $0 \leq x \rightarrow \neg cex_1$, and we have $\mathit{LE}(\hat{0}, \hat{x}) \rightarrow \neg acex_1$ after abstraction. Therefore, combining $\mathit{LE}(\hat{0}, \hat{x})$ instead of $\neg acex_1$ to $\Phi_{drl}$ can prune more redundant abstract state. The datapath refinement lemmas derived from MUSes can refute more spurious counterexamples than $\neg acex_1$. DP-Refine derives the datapath refinement lemma $drl_1 : \mathit{LE}(\hat{0}, \hat{x})$ and adds it to $\Phi_{drl}$.

In the second iteration, IC3 checks 1-step safety ($\hat{I} \wedge \hat{T} \neg \hat{P}' \wedge drl_1$) and returns a 1-step ACEX $acex_2$. We don't show the trace for brevity. The corresponding bit-level formula $cex_2$:

$$x = 0 \wedge y = 0 \wedge x \geq y \wedge x' = x + 1 \wedge y' = y + 1 \wedge y' > x'$$

is found to be infeasible by the SMT solver. Then, the refinement procedure refutes $acex_2$ by generating datapath refinement lemma $drl_2$:

$$\neg(\hat{x} = \hat{0} \wedge \hat{y} = \hat{0} \wedge \hat{x}' = \mathit{ADD}(\hat{x}, \hat{1}) \wedge \hat{y}' = \mathit{ADD}(\hat{y}, \hat{1}) \wedge \neg \mathit{LE}(\hat{y}', \hat{x}'))$$

The third iteration also returns a 1-step ACEX $acex_3$, which is found to be infeasible and refuted by datapath refinement lemma $drl_3 : \neg(\hat{y} = \hat{x} \wedge \mathit{LT}(\hat{x}, \hat{y}))$. Finally, after six refinements, DP-IC3 succeeds in finding an inductive invariant $\hat{y} = \hat{x}$ and proves that $\hat{P}$ (and $P$) holds.

## *B. Datapath Knowledge is Important*

The inherent advantage of the DP-IC3 is that the reachability computation is performed on the abstract model of the hardware design, which hides the bit-level details of datapath operations. From the angle of the SMT solver, constraint solving for UFs is much faster than when bit-level facts must be involved. Therefore, each IC3 call is expected to be more efficient than the bit-level IC3 call. However, the bit-level IC3 is only called once, but DP-IC3 may call IC3 iteratively in abstract state space because of the spurious counterexamples. Therefore, the number of CEGAR iterations is crucial to the overall verification efficiency.

Applying the knowledge of datapath operations can reduce the CEGAR iterations. Consider the 0-step safety check of the example design in Fig. 2. Since predicate *LE* is uninterpreted for SMT solver, and there are no other constraints on it, $\mathit{LE}(\hat{y}, \hat{x})$ can be assigned to any boolean values, which results in the spurious counterexample $acex_1$. Each CEGAR iteration is complex. DP-IC3 needs to generate the bit level counterexample $cex_1$, checking its feasibility, then invokes DP-Refine to generate datapath refinement lemmas that refute $acex_1$. In contrast, $0 \leq 0$ (even $0 \leq x$) is trivial for $\leq$, considering its semantics. Therefore, conveying the information that $\mathit{LE}(\hat{0}, \hat{0})$ or $\mathit{LE}(\hat{0}, \hat{x})$ to DP-IC3 is useful for avoiding the spuriousness and reducing CEGAR iterations.

Moreover, applying the knowledge datapath operations can reduce the size of the query formula. In DP-IC3, every query formula is augmented by $\Phi_{drl}$. As the size of $\Phi_{drl}$ grows along with the CEGAR iterations, the size of the query formula also grows rapidly. Therefore, reducing the CEGAR iterations can also reduce the size of the query formula in abstract space and achieves higher constraint solving efficiency.

However, the knowledge of datapath operations is neglected by DP-IC3. Datapath operations are the essential arithmetic or logical units in the design which comprise complex functionalities. Roughly treating all datapath operations as UFs causes coarse abstraction, which may bring numerous spurious counterexamples and put a heavy burden on the CEGAR framework. Instead, for a query formula $\hat{\varphi}$ in abstract state space, utilizing the knowledge of datapath operations may prune the redundant search space, reduce the number of CEGAR iterations, and improve the verification efficiency.

One may consider all the semantics of datapath operations. Then, DP-IC3 degenerates to the bit-level IC3 algorithm since no abstraction exists and it suffers from the state space explosion problem. We propose a datapath propagation procedure to take advantage of DP-IC3 and utilize the knowledge of datapath operations.

## IV. Datapath Propagation

This section details the datapath propagation. We introduce the supported datapath operations and we show the propagation rules that carry the knowledge of datapath operations. Then, we show the workflow of the propagation procedure.

Let $\varphi$ be a bit-level formula in *conjunctive normal form* (CNF), e.g., $\varphi = C_1 \wedge C_2 \wedge \dots C_n$ where $C_i, 0 \leq i \leq n$, is a clause. Denote $\alpha$ and $\gamma$ as the DP-Abstract and DP-Concrete functions in Alg. 2. Therefore, $\alpha$ replaces constants, variables, or datapath operations with UFs; $\gamma$ is just the opposite. $\alpha$ and $\gamma$ maintain the correspondence between an abstract entity and its bit-level counterpart in datapath abstraction. Let $u$ be a constant, a variable, or a CNF formula, we use $\hat{u} = \alpha(u)$ to represent its abstract version and $u = \gamma(\hat{u})$. Tab. I lists the abstract version of supported datapath operations, which can be divided into several categories:

- **Arithmetic Operations**: mathematical computations on data, including addition (+), subtraction (-), multiplication ($\times$), division (/), and modulo (%) operations.
- **Relational Operations**: data comparison operations that returns a Boolean value, including $<$ and $\leq$. We use their negation to represent $\geq$ and $>$ to reduce the types and facilitate the further analysis.
- **Bitwise Operations**: logical operations manipulate binary data using Boolean logic, including Bitwise AND (&), Bitwise OR (|), Bitwise XOR (^), Bitwise NOT ($\sim$), and negations of the first three operations.
- **Reduction Operations**: logical operations that reduce a set of data elements to a Boolean value, including reduction AND, OR, XOR, and their negation.
- **Shift Operations**: data movement operations that shift the binary representation of data to the left or right, including: logical left shift (<<) and logical right shift (>>) where the empty bits are filled with zero; arithmetic left shift (<<<) and arithmetic right shift (>>>) where the sign bit is used to fill the empty bit positions.

Let $symb(\hat{\varphi})$ be a set of symbols appearing in $\hat{\varphi}$. For example, suppose that $\hat{\varphi}$ is $\hat{x} = \hat{0} \wedge \hat{y} = \mathit{ADD}(\hat{x}, \hat{1}) \wedge \mathit{LT}(\hat{y}, \hat{x})$ we have $symb(\hat{\varphi}) = \{\hat{x}, \hat{y}, \hat{0}, \hat{1}, \mathit{LT}, \mathit{ADD}\}$.

The main idea of datapath propagation is to first recognize constant symbols in $\hat{\varphi}$ and propagate them to related UFs. We consider the original datapath operations of these UFs and try to obtain the outcomes of the original datapath operations. Moreover, the knowledge of datapath operations is not limited to the constant symbols, we also devise the propagation rules to utilize this knowledge and obtain the outcomes of related datapath operations. Then we assign the outcomes back to the corresponding UFs and continue the iterative propagation. In the following, we introduce the propagation rules.

### A. Propagation Rules

Propagation rules vary with the datapath operation's type. Let $\hat{x}, \hat{y}, \hat{z}$ be 0-ary UFs that represent abstract state variables after datapath abstraction. Denote 0-ary UFs $\hat{0}, \hat{1}, \dots$ the constant symbols. Let $MAX_x$ be $2^{x.width()} - 1$. Tab. II lists the essential propagation rules for supported datapath operations.

TABLE I
Abstract datapath operations

| Type | After DP-Abstract |
|---|---|
| Arithmetic | ADD, SUB/Minus, MUL, DIV, MOD |
| Relational | LT, LE |
| Bitwise | BitWiseAnd, BitWiseOr, BitWiseXor, BitWiseNAnd, BitWiseNor, BitWiseXNor, BitWiseNot |
| Reduction | ReductionAnd, ReductionOr, ReductionXor, ReductionNAnd, ReductionNor, ReductionXNor |
| Shift | ShiftL, ShiftR, AShiftL, AShiftR |

The first row shows the propagation rules for arithmetic operations. We take $\mathit{MUL}(\hat{x}, \hat{y}) = \alpha(x \times y)$ as an example:

- if $\hat{x}$ is not equal to any constant symbol in $symb(\hat{\varphi})$ and $\hat{y} = \hat{0}$, $\mathit{MUL}(\hat{x}, \hat{y})$ is propagated to $\hat{0}$.
- if $\hat{x}$ is not equal to any constant symbol in $symb(\hat{\varphi})$ and $\hat{y} = \hat{1}$, $\mathit{MUL}(\hat{x}, \hat{y})$ is propagated to that $\hat{x}$.

The other two symmetric cases have the same result. Moreover, $\hat{\varphi}$ is a CNF formula, i.e., $\hat{\varphi} = \hat{C_1} \wedge \hat{C_2} \wedge \dots \hat{C_n}$. If $\hat{\varphi}$ is satisfiable, both $\hat{C_i}, 0 < i \leq n$ should be satisfiable. Therefore, for some $\hat{C_i}$ that are equalities between UFs, we put these UFs into a set (called *equality closure*).

**Example1***:* For example, suppose $\hat{\varphi}$ is

$$\hat{x} = \hat{y} \wedge \hat{u} = \mathit{SUB}(\hat{x}, \hat{y}) \wedge \hat{v} = \hat{0} \wedge \mathit{LT}(\hat{u}, \hat{v}) \wedge \dots$$

we maintain equality closures $\{\hat{x}, \hat{y}\}$ and $\{\hat{v}, \hat{0}\}$ to facilitate further propagations. Consider the propagation rules about arithmetic operations in Tab. I, $\hat{x}$ and $\hat{y}$ belong to the same equality closure, then $\mathit{SUB}(\hat{x}, \hat{y})$ should be $\hat{0}$. Since $\hat{0} \in symb(\hat{\varphi})$, we have $\hat{u} = \hat{0}$, and we add $\hat{u}$ to the second equality closure. Since $\hat{u} = \hat{v}$, the propagation rule about relational operations in Tab. I can be applied. We have $\neg \mathit{LT}(\hat{u}, \hat{v})$, which contradicts the predicates $\mathit{LT}(\hat{u}, \hat{v})$ in $\hat{\varphi}$.

Note that he parameters of UFs can be other UFs, e.g., $(\hat{x} = \hat{y}) \rightarrow \mathit{LE}(\mathit{ADD}(\hat{x}, \hat{1}), \mathit{ADD}(\hat{y}, \hat{1}))$. Additionally, there are some special rules for reduction operations. For example, suppose $\hat{x}$ is determined to be unequal to $\hat{0}$ in some $\hat{C_i}$ and $\mathit{ReductionOr}(\hat{x})$ appears in $\hat{\varphi}$, we replace $\mathit{ReductionOr}(\hat{x})$ with $\hat{1}$ if $\hat{1} \in symb(\hat{\varphi})$. Therefore, the propagation builds equality between the UF and the constant symbol, which is unknown to the SMT solver.

Thirdly, there are some special rules for relational operations. Consider the second row in Tab. I, if $\mathit{LT}$ and $\mathit{LE}$ appear in $\hat{\varphi}$ and they have the same set of parameters, we have propagation rules $\mathit{LT}(\hat{x}, \hat{y}) \rightarrow \neg \mathit{LE}(\hat{y}, \hat{x})$ since $x < y$ means $\neg(y \leq x)$ and $\mathit{LE}(\hat{x}, \hat{y}) \rightarrow \neg \mathit{LT}(\hat{y}, \hat{x})$ since $x \leq y$ means $\neg(y < x)$. Moreover, $<$ and $\leq$ are transitive; Fig. 3 provides an intuitive illustration of these propagation rules. In (a), suppose that some $\hat{C_i}$ in $\hat{\varphi}$ are predicates $\mathit{LT}(\hat{x}, \hat{y})$ and $\mathit{LT}/\mathit{LE}(\hat{y}, \hat{u})$ (the solid line), and $\mathit{LT}/\mathit{LE}(\hat{x}, \hat{u})$ appears in $\hat{\varphi}$. Since $x < y \wedge y < u$ implies $x < u$ and $x < y \wedge y \leq u$ implies $x < u$, we propagate $\mathit{LT}(\hat{x}, \hat{u})$ and $\mathit{LE}(\hat{x}, \hat{u})$ to *true* (the green checkmark). On the contrary, the predicates $\mathit{LT}(\hat{u}, \hat{x})$ and $\mathit{LE}(\hat{u}, \hat{x})$ are propagated to *false* (the red cross) if they appear in $\hat{\varphi}$.

TABLE II
PROPAGATION RULES FOR ARITHMETIC, RELATIONAL, BITWISE, REDUCTION, AND SHIFT OPERATIONS

| Type | Propagation rules | | | | |
|---|---|---|---|---|---|
| Arithmetic | $ADD(\hat{x},\hat{0})=\hat{x}$ | $ADD(\hat{0},\hat{y})=\hat{y}$ | $SUB(\hat{x},\hat{0})=\hat{x}$ | $(\hat{x}=\hat{y})\rightarrow SUB(\hat{x},\hat{y})=\hat{0}$ | $MUL(\hat{x},\hat{0})=\hat{0}$ |
| | $MUL(\hat{0},\hat{y})=\hat{0}$ | $MUL(\hat{x},\hat{1})=\hat{x}$ | $MUL(\hat{1},\hat{y})=\hat{y}$ | $(\hat{x}=\hat{y})\rightarrow DIV(\hat{x},\hat{y})=\hat{1}$ | $DIV(\hat{0},\hat{y})=\hat{0}$ |
| | $MOD(\hat{x},\hat{1})=\hat{0}$ | $MOD(\hat{0},\hat{x})=\hat{0}$ | $(\hat{x}=\hat{y})\rightarrow MOD(\hat{x},\hat{y})=\hat{0}$ | | |
| Relational | $\neg LT(\hat{x},\hat{x})$ | $(\hat{x}=\hat{y})\rightarrow\neg LT(\hat{x},\hat{y})$ | $LT(\hat{x},\hat{z})\wedge LT(\hat{z},\hat{y})\rightarrow LT(\hat{x},\hat{y})$ | $LT(\hat{x},\hat{y})\rightarrow LE(\hat{x},\hat{y})$ | |
| | $\neg LT(\hat{x},\hat{0})$ | $LE(\hat{x},\hat{y})\rightarrow\neg LT(\hat{y},\hat{x})$ | $LE(\hat{x},\hat{z})\wedge LE(\hat{z},\hat{y})\rightarrow LE(\hat{x},\hat{y})$ | $LT(\hat{x},\hat{z})\wedge LE(\hat{z},\hat{y})\rightarrow LT(\hat{x},\hat{y})$ | |
| | $LE(\hat{0},\hat{x})$ | $(\hat{x}=\hat{y})\rightarrow LE(\hat{x},\hat{y})$ | $LE(\hat{x},\hat{z})\wedge LE(\hat{z},\hat{y})\rightarrow LE(\hat{x},\hat{y})$ | | |
| | $LE(\hat{x},\hat{x})$ | $LT(\hat{x},\hat{y})\rightarrow\neg LE(\hat{y},\hat{x})$ | $LE(\hat{x},\hat{z})\wedge LT(\hat{z},\hat{y})\rightarrow LT(\hat{x},\hat{y})$ | | |
| Bitwise | $BitWiseAnd(\hat{x},\hat{0})=\hat{0}$ | $BitWiseOr(\hat{x},\hat{0})=\hat{x}$ | $BitWiseNor(\hat{MAX_x},\hat{y})=\hat{0}$ | $(\hat{x}=\hat{y})\rightarrow BitWiseAnd(\hat{x},\hat{y})=\hat{x}$ | |
| | $BitWiseAnd(\hat{0},\hat{y})=\hat{0}$ | $BitWiseOr(\hat{0},\hat{y})=\hat{y}$ | $BitWiseOr(\hat{x},\hat{MAX_x})=\hat{MAX_x}$ | $(\hat{x}=\hat{y})\rightarrow BitWiseOr(\hat{x},\hat{y})=\hat{x}$ | |
| | $BitWiseAnd(\hat{x},\hat{MAX_x})=\hat{x}$ | $BitWiseOr(\hat{MAX_x},\hat{y})=\hat{MAX_x}$ | $BitWiseNAnd(\hat{x},\hat{0})=\hat{MAX_x}$ | $(\hat{x}=\hat{y})\rightarrow BitWiseXor(\hat{x},\hat{y})=\hat{0}$ | |
| | $BitWiseAnd(\hat{MAX_x},\hat{y})=\hat{y}$ | $BitWiseNor(\hat{x},\hat{MAX_x})=\hat{0}$ | $BitWiseNAnd(\hat{0},\hat{y})=\hat{MAX_x}$ | $(\hat{x}=\hat{y})\rightarrow BitWiseXNor(\hat{x},\hat{y})=\hat{MAX_x}$ | |
| Reduction | $ReductionAnd(\hat{x})=\hat{0}$ where $\neg(\hat{x}=\hat{MAX_x})$ | $ReductionOr(\hat{x})=\hat{1}$ where $\neg(\hat{x}=\hat{0})$ | $ReductionNOr(\hat{x})=\hat{1}$ where $\neg(\hat{x}=\hat{0})$ | | |
| | $ReductionNAnd(\hat{x})=\hat{0}$ where $\neg(\hat{x}=\hat{MAX_x})$ | $ReductionXNOr(\hat{x})=\hat{1}$ where $\neg(\hat{x}=\hat{MAX_x})$ | | | |
| Shift | $ShiftL(\hat{x},\hat{0})=\hat{x}$ | $ShiftL(\hat{0},\hat{x})=\hat{0}$ | $ShiftR(\hat{x},\hat{0})=\hat{x}$ | $ShiftR(\hat{0},\hat{x})=\hat{0}$ | |
| | $AShiftL(\hat{x},\hat{0})=\hat{x}$ | $AShiftL(\hat{0},\hat{x})=\hat{0}$ | $AShiftR(\hat{x},\hat{0})=\hat{x}$ | $AShiftR(\hat{0},\hat{x})=\hat{0}$ | |

In contrast, consider $LE(\hat{x},\hat{y})$ and $LE(\hat{y},\hat{u})$ in (b). Since $x\le y\wedge y\le u$ implies $x\le u$, we propagate $LE(\hat{x},\hat{u})$ to *true* (the green checkmark) and $LT(\hat{u},\hat{x})$ just the opposite (the red cross). Note that $LT(\hat{x},\hat{u})$ is still unknown if it appears in $\hat{\varphi}$. Moreover, since $x\le y\wedge y<z$ implies $x<z$, the bottom half of (b) has the same propagation results as (a).

## B. Propagation Procedure

Given an abstract CNF formula $\hat{\varphi}$, *datapath propagation* is a procedure that tries to construct a formula $\hat{\psi}$, s.t., $\hat{\varphi}\wedge\hat{\psi}$ is *unsat* and $\models_{\mathcal{T}}\psi$ where $symb(\hat{\psi})\subseteq symb(\hat{\varphi})$ and $\psi=\gamma(\hat{\psi})$. In more detail, $\hat{\psi}$ is a formula over UFs and $\psi=\gamma(\hat{\psi})$ its bit-level counterpart. $\models_{\mathcal{T}}\ \psi$ means $\psi$ is a tautology under first order theory $\mathcal{T}$, e.g., bit-vector theory. Datapath propagation tries to find a formula $\hat{\psi}$ over UFs that $\hat{\varphi}\wedge\hat{\psi}$ is unsatisfiable. $symb(\hat{\psi})\subseteq symb(\hat{\varphi})$ indicates that all the symbols appeared in $\hat{\psi}$ also appear in $\hat{\varphi}$, i.e., $\hat{\psi}$ does not introduce new uninterpreted constants, variables, predicates, or functions that are not in $symb(\hat{\varphi})$.

**Example2***:* Let $\hat{\varphi}$ be $\hat{x}=\hat{0}\wedge\hat{y}=ADD(\hat{x},\hat{1})\wedge LT(\hat{y},\hat{x})$, a possible assignment returned by the SMT solver is:

$$\{\hat{x}\mapsto\hat{0},\ \hat{y}\mapsto v_1,\ ADD(\hat{x},\hat{1})\mapsto v_1\ LT(\hat{y},\hat{x})\mapsto \mathit{true}\}$$

where $v_1$ can be any uncertainty value that does not exceed the maximum value in its bit width. In contrast, datapath propagation may find a formula $\hat{\psi}: ADD(\hat{0},\hat{1})=\hat{1}\wedge\neg LT(\hat{1},\hat{0})$. It is easy to see that $\hat{\varphi}\wedge\hat{\psi}$ is *unsat* and the bit-level formula $\psi: 0+1=1\wedge\neg(1<0)$ is tautology under BV theory. Meanwhile, $symb(\hat{\psi})\subseteq symb(\hat{\varphi})$.

Datapath propagation is restricted to only adding constants/symbols or function composition terms appeared in $symb(\hat{\varphi})$. The reason for this restriction is straightforward. Firstly, if the propagation adds a new symbol $s$ that is not in $symb(\hat{\varphi})$, $s$ is treated as a free variable and has no contributions to prune the abstract state space. Secondly, introducing new symbols may lead to a longer search of the propagation. For example, suppose a new constant symbol $\hat{2}$ is introduced by ADD$(\hat{1},\hat{1})$ and $\hat{2}$ is not appeared in $symb(\hat{\varphi})$. Then, the propagation procedure may generate $\hat{3}$ by ADD$(\hat{2},\hat{1})$ and $\hat{4},\hat{5}\ldots$, which leads to redundant propagation and generates numerous useless formulas. Thirdly, creating function composition terms that are not in $symb(\hat{\varphi})$ is useless; and it leads to a longer search of the propagation. Finally, introducing new uninterpreted constants or functionally composed UFs to the vocabulary without any restrictions may lead to divergence in the word-level IC3 algorithm within the abstract space.

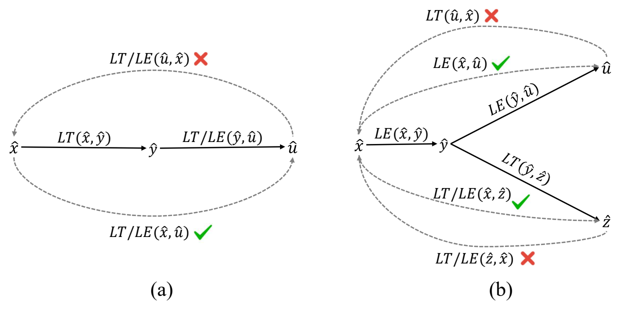

Fig. 3. Propagation rules about relational operations

For each query formula, datapath propagation terminates in three situations: 1) success in finding formula $\hat{\psi}$; 2) no more propagation can be conducted; 3) failure to find $\hat{\psi}$ within the specific number of iterations. For the first situation, since $\psi$ is a bit-level fact and $\hat{\varphi}\wedge\hat{\psi}$ is unsatisfiable, $\hat{\psi}$ is a datapath lemma that eliminates spurious ACEX. In the last two situations, no more propagation can be applied, or the size of $\hat{\varphi}$ is too big to finish the propagation, then $\hat{\varphi}$ is passed to the SMT solver.

**Algorithm 3** Datapath propagation $(\hat{\varphi}, bound, \Phi_{dpl})$

```
1: $\hat{\varphi}_p = \hat{\varphi},\ \hat{\varphi}_q = true,\ k = 0,\ \hat{\psi} = true$;
2: while $\hat{\varphi}_p\ != \ \hat{\varphi}_q$ and $k < bound$ do
3:     $\hat{\varphi}_q = \hat{\varphi}_p$;
4:     let $\hat{C}$ be constant symbols in $\hat{\varphi}_p$;
5:     for $\hat{c} \in \hat{C}$ do
6:         $UF_c$ = update_related_UF($\hat{c}, \hat{\varphi}_p$);
7:         for $uf \in UF_c$ do
8:             if parameters are all constant symbols in $uf$ then
9:                 $res = \gamma(uf),\ \hat{res} = \alpha(res)$;
10:                if $\hat{res} \in symb(\hat{\varphi}_p)$ then
11:                    $\hat{\varphi}_p$ = replace_UF_with_const($\hat{\varphi}_p, uf, \hat{res}$);
12:                    $\hat{\psi} = \hat{\psi} \wedge (uf = \hat{res})$;
13:                else
14:                    $\hat{\psi} = \hat{\psi} \wedge \neg(uf = \hat{u})$ foreach $\hat{u} \in \hat{C}$;
15:            else
16:                $\hat{\psi}, \hat{\varphi}_p$ = apply_propagation_rule($uf$);
17:            if $\hat{\varphi}_p \wedge \hat{\psi}$ is unsat then
18:                $\Phi_{dpl} = \Phi_{dpl} \wedge \hat{\psi}$;
19:                return unsat;
20:    $\hat{\psi}, \hat{\varphi}_p$ = apply_propagation_rule($\hat{\varphi}_p$);
21:    if $\hat{\varphi}_p \wedge \hat{\psi}$ is unsat then
22:        $\Phi_{dpl} = \Phi_{dpl} \wedge \hat{\psi}$;
23:        return unsat;
24:    $k = k + 1$;
25: $\Phi_{dpl} = \Phi_{dpl} \wedge \hat{\psi}$;
26: return unknown;
```

However, some bit-level facts $\lambda$ can also be propagated in the last two situations. $\hat{\lambda} = \alpha(\lambda)$ is a datapath lemma that reduces the abstract state space and facilitates further verification. We call $\hat{\psi}$ or $\hat{\lambda}$ *datapath propagation lemma* (DPL).

Alg. 3 details the datapath propagation. The inputs include an abstract query formula $\hat{\varphi}$, a maximal propagation depth *bound*, and a formula $\Phi_{dpl}$ that records datapath propagation lemmas (DPLs). The algorithm tries to construct a formula $\hat{\psi}$ that $\models_{\mathcal{T}} \psi$ and $\hat{\varphi} \wedge \hat{\psi}$ is *unsat*. It returns *unsat* if successful, or *unknown* in other situations. First, we set two formulas $\hat{\varphi}_p = \hat{\varphi}$ and $\hat{\varphi}_q = true$. They are used to check if propagation continues. $k = 0$ is the current depth of the propagation, and $\hat{\psi}$ is initially assigned *true* (line 1).

The while loop in lines 2-24 is the main procedure, which exits when $\hat{\varphi}_p = \hat{\varphi}_q$ or $k \geq bound$ (line 2). Since $\hat{\varphi}_q$ is the query formula before executing the loop and $\hat{\varphi}_p$ is the query formula after the loop, $\hat{\varphi}_p = \hat{\varphi}_q$ means no propagation is conducted in the last iteration. $k \geq bound$ means the number of iterations exceeds the specific depth. In the loop, we first assign $\hat{\varphi}_p$ to $\hat{\varphi}_q$ (line 3) and let $\hat{C}$ be the set of constant symbols in $\hat{\varphi}$ (line 4).

For each constant symbol $\hat{c} \in \hat{C}$, we propagate $\hat{c}$ to related UFs in $\hat{\varphi}$ and use $UF_c$ to collect the updated UFs (line 6). In more detail, suppose that $\hat{\varphi}$ is

$$\hat{x} = \hat{0} \wedge \hat{y} = \hat{x} \wedge \hat{z} = \hat{1} \wedge \hat{v} = ADD(\hat{u}, \hat{y}) \wedge LT(\hat{x}, \hat{z})$$

propagating $\hat{0}$ to UFs in $\hat{\varphi}$ returns $\hat{z} = \hat{1} \wedge \hat{v} = ADD(\hat{u}, \hat{0}) \wedge LT(\hat{0}, \hat{z})$. Then, $ADD(\hat{u}, \hat{0})$ and $LT(\hat{0}, \hat{z})$ are added to $UF_c$.

For each $uf \in UF_c$ (line 7), if all the parameters of $uf$ are constant symbols; the algorithm considers the original datapath operation $\gamma(uf)$ and tries to obtain its outcome. Let *res* represents the outcome and $\hat{res} = \alpha(res)$ its abstract version (line 9). If $\alpha(res)$ exists in $symb(\hat{\varphi})$, we update $\hat{\varphi}$ by replacing *uf* with *res* (line 11). Moreover, we build an equality between $\hat{res}$ and *uf* and combines the equality into $\hat{\psi}$ (line 12). However, if $\hat{res}$ introduces new symbol that does not exist in $symb(\hat{\varphi})$, we combine inequalities $\neg(uf = \hat{u})$ into $\hat{\psi}$ for each $\hat{u}$ in $C$ (line 14). For example, suppose $ADD(\hat{x}, \hat{y})$ appears in $\hat{\varphi}$ and we know that $\hat{x} = \hat{1}$ and $\hat{y} = \hat{1}$. Since $1 + 1 = 2$, we check if $\hat{2} \in symb(\hat{\varphi})$. If so, we replace $ADD(\hat{x}, \hat{y})$ with $\hat{2}$ in $\hat{\varphi}$ and combine $ADD(\hat{1}, \hat{1}) = \hat{2}$ into $\hat{\psi}$. If only $\hat{0}, \hat{1}$, and $\hat{3}$ appear in $\hat{\varphi}$, then we combine $\neg(ADD(\hat{1}, \hat{1}) = \hat{0})$, $\neg(ADD(\hat{1}, \hat{1}) = \hat{1})$, and $\neg(ADD(\hat{1}, \hat{1}) = \hat{3})$ into $\hat{\psi}$.

If at least one parameter is not a constant symbol in *uf* (line 15), we may not obtain the value outcome of $\gamma(uf)$. In this situation, we apply propagation rules described in the last section, which consider the original semantics of supported datapath operations (line 16). If the updated formula $\hat{\varphi} \wedge \hat{\psi}$ is *unsat*, $\hat{\psi}$ is enough to demonstrate that $\varphi$ is $\hat{\mathcal{T}}$-unsatisfiable. Then, the algorithm combines $\hat{\psi}$ to $\Phi_{dpl}$ and returns *unsat* (lines 17-19). Otherwise, the algorithm continues to pick the next $uf \in UF_c$ (line 7). If all the UFs in $UF_c$ are processed, the algorithm continues to operate the next constant symbol in $\hat{C}$ (line 5). In summary, the nested loop from lines 5 to 19 propagates constant symbols to related UFs.

Next, Alg. 3 performs propagations beyond constant symbols using propagation rules in Tab. II (line 20). If $\hat{\varphi} \wedge \hat{\psi}$ is *unsat*, the algorithm combines $\hat{\psi}$ to $\Phi_{dpl}$ and returns *unsat* (lines 21-23); Otherwise, $k = k + 1$ and it continues the next round of datapath propagation (line 24). Finally, if $\hat{\varphi}_p = \hat{\varphi}_q$, i.e., no more propagations can be conducted, or $k$ exceeds the maximal depth, Alg. 3 combines $\hat{\psi}$ to $\Phi_{dpl}$ and returns *unknown* (lines 25-26).

Datapath propagation iteratively utilizes the above propagation rules and tries to obtain the outcomes of related UFs. We have the following theorem:

*Theorem 1: Datapath propagation is sound and incomplete.*

The theorem is straightforward. Propagation rules are originated from the original semantics of datapath operations. Datapath propagation applies these rules to UFs, which is apparently a sound procedure. For completeness, since we only support part of datapath operations and iterations are not exhaustive, datapath propagation is an incomplete procedure.

## V. Applying Datapath Propagation in DP-IC3

Fig. 4 shows a high-level overview of the DP-IC3 with datapath propagation. Taking a model checking problem $\mathcal{P} := \langle X, I, T, P \rangle$ as input, we obtain $\hat{\mathcal{P}}$ after DP-Abstract and initialize $\Phi_{dpl}$ and $\Phi_{drl}$ to *true*, which record the datapath propagation lemmas and datapath refinement lemmas, respectively. Both $\Phi_{dpl}$ and $\Phi_{drl}$ are global lemmas that apply to all queries in the current (and future) abstract space.

IC3 is performed in the abstract space. An abstract query formula $\hat{\varphi}$ encodes a top frame query $\hat{F}_k \wedge \hat{T} \wedge \neg\hat{P}$ or an

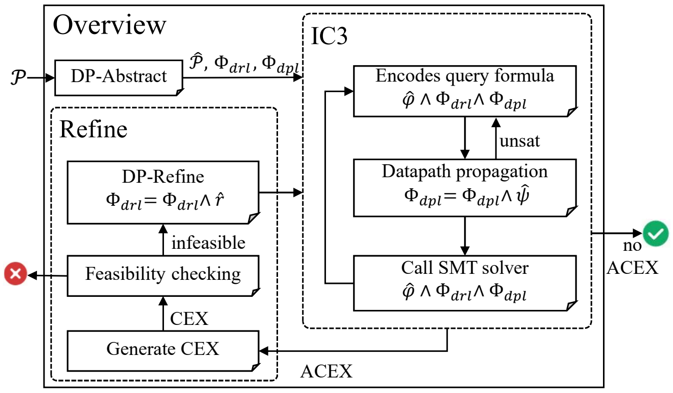
Overview
IC3
$\mathcal{P}$
DP-Abstract
$\hat{\mathcal{P}}, \Phi_{drl}, \Phi_{dpl}$
Encodes query formula
$\hat{\varphi} \wedge \Phi_{drl} \wedge \Phi_{dpl}$
unsat
Datapath propagation
$\Phi_{dpl} = \Phi_{dpl} \wedge \hat{\psi}$
Call SMT solver
$\hat{\varphi} \wedge \Phi_{drl} \wedge \Phi_{dpl}$
no ACEX
Refine
DP-Refine
$\Phi_{drl} = \Phi_{drl} \wedge \hat{r}$
infeasible
Feasibility checking
CEX
Generate CEX
ACEX

Fig. 4. Overview of the DP-IC3 with datapath propagation

intermediate query in the IC3 algorithm. For each query $\hat{\varphi}$, IC3 encodes the query formula $\hat{\varphi} \wedge \Phi_{dpl} \wedge \Phi_{drl}$ and invokes datapath propagation. After the propagation, IC3 combines the newly-generated DPL $\hat{\psi}$ to $\Phi_{dpl}$. If the propagation returns *unsat*, IC3 continues to encode the next abstract query formula as needed. Otherwise, the algorithm passes the query formula $\hat{\varphi} \wedge \Phi_{dpl} \wedge \Phi_{drl}$ to the SMT solver. Then, IC3 continues until either it returns an empty trace indicating that $\hat{P}$ holds (the green checkmark) or it returns a non-empty trace ACEX.

ACEX is passed to the refinement procedure. It first generates the bit-level counterpart CEX and performs feasibility checking. If CEX is feasible, a real counterexample is found, and $P$ is violated (the red cross). Otherwise, CEX is BV-unsatisfiable, DP-Refine generates datapath refinement lemmas $\hat{r}$ and combines $\hat{r}$ to $\Phi_{drl}$ to eliminate ACEX. Then, the verification framework calls the next round of IC3.

Compared to DP-IC3, which puts all the burden of tightening datapath abstraction on refinement's shoulders, applying datapath propagation has two main advantages. Firstly, we convey the knowledge of datapath propagations to abstract state space by assigning some UFs with accurate values or building the relation between some UFs. This knowledge is unknown to the SMT solver but can avoid spurious counterexamples. Secondly, we generate DPLs during the datapath propagation. Since datapath propagation is independent of CEGAR, combining our method with DP-IC3 can further prune the abstract state space and reduce the number of CEGAR iterations. The verification efficiency is thus improved.

We develop some strategies to make datapath propagation a lightweight and fast procedure. First, the query formula $\hat{\varphi} = \hat{F}_k \wedge \hat{T} \wedge \neg\hat{P}'$ may be called many times in IC3 (line 6 in Alg. 1). Therefore, for each abstract query formula $\hat{\varphi} \wedge \hat{\Phi}_{drl} \wedge \hat{\Phi}_{dpl}$, if no DPL is generated from the current propagation and no new syntax term is introduced in refinement, we skip the datapath propagation for its next query. Second, we want more concise datapath lemmas. If the size of a query formula is too big, the generated DPL may be too long and thus too weak to eliminate spuriousness. Therefore, we set a maximal propagation depth *bound* (currently 20) to limit the search depth. Actually, in most cases, the datapath propagation terminates within *bound* for each abstract query formula $\hat{\varphi}$; either because it finds DPL $\hat{\psi}$ that $\hat{\varphi} \wedge \hat{\psi}$ is *unsat*, or no more propagations can be conducted.

**Example3***:* Considering the example design in Fig. 2, we show how the datapath propagation works with the DP-IC3 algorithm. First, the 0-step abstract query formula $\hat{\varphi} = \hat{I} \wedge \neg\hat{P}$, i.e., $\hat{x} = \hat{0} \wedge \hat{y} = \hat{0} \wedge \neg LE(\hat{y}, \hat{x})$. If $\hat{\varphi}$ is satisfiable, $\hat{x} = \hat{0}$ and $\hat{y} = \hat{0}$ should satisfiable too. Therefore, we propagate $\hat{0}$ to $LE(\hat{y}, \hat{x})$ and obtain $LE(\hat{0}, \hat{0})$. Since $\models_{\mathcal{T}} 0 \leq 0$, we have $LE(\hat{0}, \hat{0})$ is *true*. Consequently, its negation is propagated to *false*. Datapath propagation returns *unsat* for $\hat{\varphi}$ and generate DPL $LE(\hat{0}, \hat{0})$. Additionally, since $\hat{x}$ and $\hat{y}$ are in the same equality closure, we obtain another DPL $\hat{x} = \hat{y} \rightarrow LE(\hat{y}, \hat{x})$.

The 1-step abstract query formula $\hat{\varphi} = \hat{I} \wedge \hat{T} \wedge \neg\hat{P}$:

$$\hat{x} = 0 \wedge \hat{y} = 0 \wedge \hat{x}' = LT(\hat{x}, \hat{y})?\hat{x} : (\hat{y}! = \hat{x})?\hat{y} : ADD(\hat{x}, \hat{1}) \wedge$$
$$\hat{y}' = (\hat{y} == \hat{x})?ADD(\hat{y}, \hat{1}) : LT(\hat{x}, \hat{y})?\hat{y} : \hat{x} \wedge \neg LE(\hat{y}', \hat{x}')$$

First, the literals $\hat{x} = \hat{0}$ and $\hat{y} = \hat{0}$ are deduced $true$. Then, $\hat{0}$ is propagated to related UFs. Since $\models_{\mathcal{T}} \neg(0 < 0)$, $LT(\hat{x}, \hat{y})$ is propagated to *false*. Meanwhile, $\hat{y}! = \hat{x}$ is also *false*. $\hat{\varphi}$ is updated to $\hat{x}' = ADD(\hat{0}, \hat{1}) \wedge \hat{y}' = ADD(\hat{0}, \hat{1}) \wedge \neg LE(\hat{y}', \hat{x}')$. Next, since $\models_{\mathcal{T}} 0 + 1 = 1$ and $\hat{1} \in symb(\hat{\varphi})$, we replace $ADD(\hat{x}, \hat{1})$ with $\hat{1}$, and we get DPL $ADD(\hat{0}, \hat{1}) = \hat{1}$. $\hat{\varphi}$ is updated to $\hat{x}' = \hat{1} \wedge \hat{y}' = \hat{1} \wedge \neg LE(\hat{y}', \hat{x}')$. Since $\models_{\mathcal{T}} 1 \leq 1$, $LE(\hat{y}', \hat{x}')$ is propagated to $true$. Meanwhile, we get DPLs $LE(\hat{1}, \hat{1})$ and $\hat{x}' = \hat{y}' \rightarrow LE(\hat{y}', \hat{x}')$. Finally, $\neg LE(\hat{y}', \hat{x}')$ is deduced to $fasle$, datapath propagation finds $\hat{\psi}$ that $\hat{\varphi} \wedge \hat{\psi}$ is *unsat* and returns *unsat* for $\hat{\varphi}$. Therefore, datapath propagation reports *unsat* to DP-IC3 and combines $\hat{\psi}$ to $\Phi_{dpl}$.

The next abstract query formula $\hat{\varphi} = \hat{P} \wedge \hat{T} \wedge \neg\hat{P}'$. Since $y \leq x \models_{\mathcal{T}} \neg(x < y)$, $LT(\hat{x}, \hat{y})$ is propagated to *false*. Then, no more propagations can be conducted because $\hat{\varphi}_p = \hat{\varphi}_q$ after the first iteration in Alg. 3. Therefore, $\hat{\varphi} \wedge \Phi_{drl} \wedge \Phi_{dpl}$ is passed to SMT solver. Finally, an inductive invariant $\hat{y} = \hat{x}$ is found after a few rounds of SMT queries. To verify the example design, DP-IC3 invokes six refinements in the CEGAR framework, but applying datapath propagation generate DPL $\hat{\psi}$ beyond CEGAR and does not call refinement.

## VI. Evaluation

This section reports the experimental results and detailed analysis with state-of-the-art verification tools.

### A. Implementation and Setup

We implemented our approach in AVR with around 8K lines C++ codes[1]. We integrate the datapath propagation and lemma generation procedures into the original verification framework. Our implementation is called AVR$_{\text{dp}}$. We compare AVR$_{\text{dp}}$ with recent well-known hardware verification tools:

- AVR[2]: a tool that implements the IC3-style reachability checking; It is the champion tool of the latest hardware model checking competition (HWMCC 2020)[3].

[1]Artifact is available: https://doi.org/10.5281/zenodo.7333164
[2]https://github.com/aman-goel/avr/commit/dbc3371
[3]https://fmv.jku.at/hwmcc20/

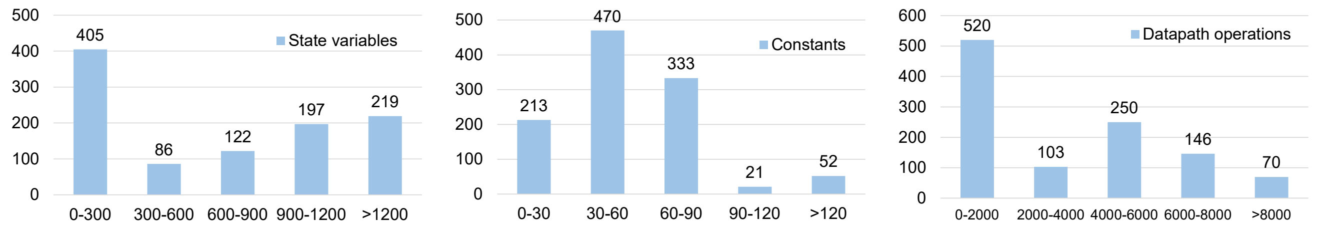

Fig. 5. The number of state variables, constants, and datapath operations of 1089 benchmarks.

- Pono (also known as CoSA2)[4]: an SMT-based model checker that implements various reachability checking techniques. It is the champion tool of HWMCC 2019[5].
- IC3IA[6]: a tool that implements implicit predicate abstraction. It performs reachability checking at the boolean level of the abstract state and eliminates spurious counterexamples by adding a sufficient set of new predicates.
- ABC[7]: a famous and widely applied system for sequential logic synthesis and formal verification. It implements the IC3-style reachability checking.
- nuXmv-ic3[8]: a recent variant of nuXmv with branching and refer-skipping heuristics [20]. It is a symbolic model checker for analyzing synchronous finite-state and infinite-state systems.
- Avy[9]: an AIGER model checker that combines interpolation and property directed reachability.

For each verification task, AVR$_{dp}$ and AVR runs in 8 default configurations, namely, `default, sa, split, sa8, sa16, sa32, level0, level5`. Pono runs in 4 default configurations, namely, `ic3sa, mbic3, ind, sygus-pdr`, and IC3IA runs in a 1 default configuration, `pa`. ABC runs in 2 configurations, `pdr` and `ind`, nuXmv runs in 5 configurations, namely, `ic3, ic3-heuristics, aic3, ind, itp`, and Avy runs in its HWMCC configuration. We omit the details of these configurations for brevity. A tool is deemed capable of solving a verification task if any of its configurations can do so. In cases where multiple configurations can yield a solution for a verification task, the fastest one is reported as the result. Such a *portfolio/proof-race* strategy is essential to achieve rapid verification performance because, generally, no single technique excels in all scenarios.

We collect all the verification tasks of the last two HWMCCs as benchmarks. We first attain 1089 word-level benchmarks for AVR$_{dp}$, AVR, Pono, and IC3IA. Fig. 5 shows the details of the scale of the benchmarks, including the number of state variables, constants, and datapath operations. Consider the first bar in the first histogram, indicating 405 cases with 0-300 state variables. The last bar in the third histogram shows 70 cases with more than 8000 datapath operations. All the word-level benchmarks are written in BTOR2 [25] format, an intermediate language for verification, and can be synthesized from Verilog by the Yosys [26] tool-chain. Note that IC3IA only supports VMT format, an extension of SMT-LIBv2; we utilize vmt-tools [10] to translate BTOR2 files into VMT files. For AVR and Pono, we use 1089 BTOR2 files as input. For IC3IA, we use 1089 VMT files as input. As for ABC, nuXmv-ic3, and Avy, we attain all 536 HWMCC 2019-2020 benchmarks in AIGER format [27].

All the experiments are conducted on a server with AMD EPYC 7H12 128-core CPU and 1 TB memory, and the operating system is Ubuntu 20.04 LTS. Following the competition, the timeout for each verification task is set to 3600 seconds.

## B. Overall Experimental Results

Tab. III summarizes the results of the above tools and AVR$_{dp}$ on all the benchmarks. Columns `Total` and `Unknown` list the number of collected benchmarks that exceed the time limit or reported parse error, respectively. Columns 4-8 display the data about the verified cases, where `Num` is the number of verified cases, and `CPU-Time` is the accumulated wall clock time, `Safe` and `Unsafe` are the number of cases that satisfy or violate the specified safety property. The last three columns display statistics of tasks that can be verified by the listed tool and AVR$_{dp}$. The speedup greater than 1.0x means that AVR$_{dp}$ is faster than the selected tool.

There are 1089 BTOR2 benchmarks in total. AVR verifies 706 cases in 57483.4 seconds, and AVR$_{dp}$ verifies 793 cases in 37918.9 seconds – AVR$_{dp}$ verifies 87 more cases and achieves 1.52x speedup than AVR. Both AVR and AVR$_{dp}$ can verify 689 cases. Considering these cases, AVR spends 38723.1 seconds whereas AVR$_{dp}$ costs 26538.0 seconds – AVR$_{dp}$ is 1.46x times faster than AVR to verify these same cases. The third row displays the comparative results of Pono and AVR$_{dp}$. Pono verifies 313 cases in 11051.7 seconds and times out for 776 cases. Considers 295 both-verified cases, Pono costs 7164.3 seconds whereas AVR$_{dp}$ only cost 3567.9 seconds – AVR$_{dp}$ is 2.01x faster than Pono. The fourth row of Tab. III shows the comparative results of IC3IA and AVR$_{dp}$. IC3IA verifies 313 tasks in 121810.1 seconds and timeout for 771 tasks. Considering the 235 both-verified cases, IC3IA spends 98675.7 whereas AVR$_{dp}$ only costs 4829.8 seconds – AVR$_{dp}$ is 20.04x faster than IC3IA.

Among 536 AIGER benchmarks, AVR$_{dp}$ verifies 345 cases in 20592.8 seconds, ABC verifies 153 cases in 7898.2 seconds, nuXmv-ic3 verifies 363 cases in 66817.7 seconds, and Avy verifies 96 cases in 24871.6 seconds. Consider 108 cases that can be verified by AVR$_{dp}$ and ABC, ABC costs 6597.1 seconds whereas AVR$_{dp}$ costs 3154.2 seconds – AVR$_{dp}$ is 2.09x faster

[4]https://github.com/upscale-project/pono/commit/b243ce
[5]https://fmv.jku.at/hwmcc19/
[6]https://es-static.fbk.eu/people/griggio/ic3ia/index.html
[7]https://github.com/berkeley-abc/abc/6ca7eab
[8]https://github.com/youyusama/i-Good_Lemmas_MC/tree/master
[9]https://arieg.bitbucket.io/avy/
[10]http://es-static.fbk.eu/people/griggio/ic3ia/vmt-tools-latest.tar.gz

TABLE III
SUMMARY OF EXPERIMENTAL RESULTS

| Verifier | Total | Unknown | Verified | | | | Both Verified | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | Num | CPU-Time (s) | Safe | Unsafe | Num | CPU-Time (s) -/AVR$_{dp}$ | Speedup |
| AVR$_{dp}$ | 1089 | 296 | 793 | 37918.8 | 762 | 31 | - | - | - |
| AVR | 1089 | 383 | 706 | 57483.4 | 675 | 31 | 689 | 38723.1/26538.0 | 1.46x |
| Pono | 1089 | 776 | 313 | 11051.7 | 301 | 12 | 295 | 7164.3/3567.9 | 2.01x |
| IC3IA | 1089 | 776 | 313 | 121810.1 | 286 | 27 | 235 | 98675.7/4829.8 | 20.40x |
| AVR$_{dp}$ | 536 | 191 | 345 | 20592.8 | 314 | 31 | - | - | - |
| ABC | 536 | 383 | 153 | 7898.2 | 99 | 54 | 108 | 6597.1/3154.2 | 2.09x |
| nuXmv-ic3 | 536 | 173 | 363 | 66817.7 | 308 | 54 | 290 | 41184.9/14506.2 | 2.84x |
| Avy | 536 | 440 | 96 | 24871.6 | 77 | 19 | 80 | 14651.2/4014.8 | 3.65x |

than ABC. Consider 290 cases that can be verified by AVR$_{dp}$ and nuXmv-ic3, nuXmv-ic3 costs 41184.9 seconds, and that number of AVR$_{dp}$ is 14506.2 – AVR$_{dp}$ is 2.84x faster than nuXmv-ic3. Compared to Avy on 80 both-verified cases, Avy and AVR$_{dp}$ cost 14651.2 and 4014.8 seconds, respectively – AVR$_{dp}$ is 3.65x faster than Avy on these cases.

## C. Results Analysis

Fig. 6 shows the number of verified cases of AVR$_{dp}$ and the comparison tool. According to (a), 104 cases can only be verified by AVR$_{dp}$. Our approach utilizes datapath propagation to prune abstract state space and generates at least one datapath lemma for each of them to guide the verification procedure. 17 cases are just the opposite; AVR$_{dp}$ is inferior to AVR on these cases. The inferiority is because the abstract query formulas are too big in these cases, and no useful datapath lemma is generated in the propagation procedure.

Fig. 6 (b) displays the number of verified cases of AVR$_{dp}$ and Pono. 498 cases can only be verified by AVR$_{dp}$. Meanwhile, there are 18 cases that AVR$_{dp}$ is inferior to Pono. The k-induction-based verification engine of Pono outperformsAVR$_{dp}$ and AVR on these cases. Fig. 7 displays the comparison results of AVR$_{dp}$ and Pono on 295 both-verified cases. Each point in the panel corresponds to a verification task, with the $X$ and $Y$ coordinates representing the `CPU-Time` of Pono and AVR$_{dp}$, respectively. Note that both $x$- and $y$-axis take logarithmic coordinates, and each point below/above the diagonal line represents a superior/inferior case of our approach against Pono. When the cases become complex, our method starts to show its strength.

Compared to IC3IA, 558 cases can only be verified by AVR$_{dp}$. IC3IA times out on 553 cases and throws exceptions on the other 5 cases. Note that IC3IA is also time-consuming for the 313 verified cases. This is because IC3IA employs predicate abstraction, which performs IC3 on the boolean level of abstract state space. It needs to learn a sufficient set of predicates to tighten the abstraction. However, generating predicates, especially useful predicates, is not easy. It may require numerous CEGAR iterations, and maintaining so many predicates is also a heavy burden for the verification procedure. Therefore, IC3IA is often trapped in situations where many arithmetic or bitwise operations are involved. On the contrary, AVR$_{dp}$ is inferior to IC3IA on 78 cases since some crucial

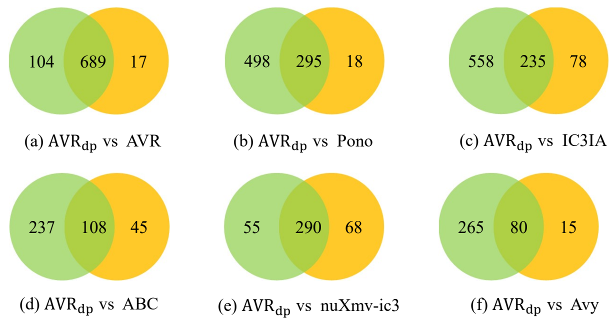

Fig. 6. The number of verified cases of AVR$_{dp}$ (the green circle) and the comparison tool (the yellow circle) where the intersecting region represents cases that both tools can verify.

predicates that witness the violation of safety property are found by IC3IA. Fig. 8 shows the comparison results of IC3IA and AVR$_{dp}$ on 235 both-verified cases. In most cases, our approach is more efficient than IC3IA.

Fig. 6 (d)-(f) show the number of verified cases of AVR$_{dp}$ and the comparison tools that take AIGER benchmarks as input. Compared to ABC, 237 cases can only be verified by AVR$_{dp}$. Meanwhile, there are 45 cases that AVR$_{dp}$ is inferior to ABC. Compared to nuXmv-ic3, 55 cases can only be verified by AVR$_{dp}$, whereas nuXmv-ic3 times out. However, there are 68 cases that AVR$_{dp}$ is inferior to nuXmv-ic3. AVR$_{dp}$ fails to generate useful datapath propagation lemmas on these cases because of numerous concat and extract operations. Compared to Avy, 265 cases can only be verified by AVR$_{dp}$, and the counterpart is 15 cases.

Note that AVR is the latest champion tool in HWMCC, i.e., it is already superior to Pono and IC3IA. Moreover, we implement the datapath propagation and DPL generation in AVR. Therefore, we take AVR as a baseline and compare further to show that our approach is effective and efficient. Fig. 9 displays the comparative results of AVR and AVR$_{dp}$ on 689 both-verified cases. The points below the diagonal represent the cases that AVR$_{dp}$ achieves higher efficiency than AVR.

Among 689 both-verified cases, there are 372 cases on which AVR$_{dp}$ generates at least one DPL in the propagation for each case and 3923 DPLs in total. AVR eliminates spurious counterexamples and tightens the datapath abstraction by refinement. Instead, AVR$_{dp}$ can generate DPL during the

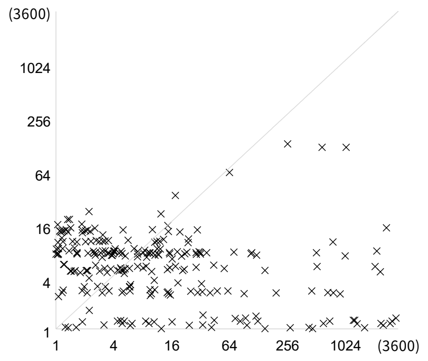

Fig. 7. Pono ($X$-axis) vs. AVR$_{\rm dp}$ ($Y$-axis) in terms of CPU-Time for each task.

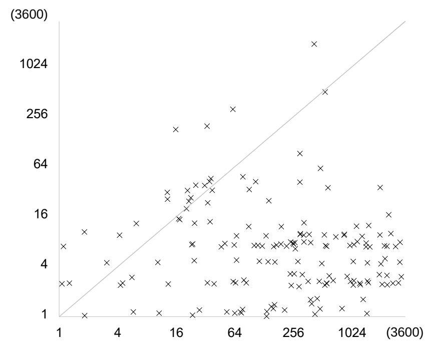

Fig. 8. IC3IA ($X$-axis) vs. AVR$_{\rm dp}$ ($Y$-axis) in terms of CPU-Time for each task.

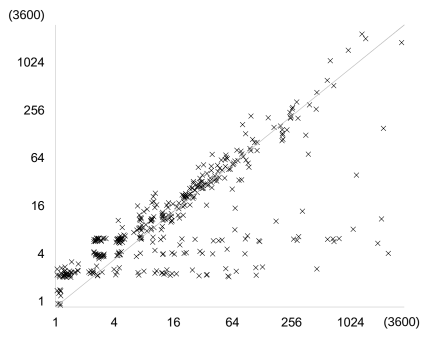

Fig. 9. AVR ($X$-axis) vs. AVR$_{\rm dp}$ ($Y$-axis) in terms of CPU-Time for each task.

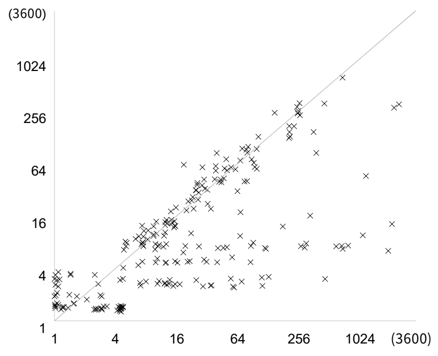

Fig. 10. AVR ($X$-axis) vs. AVR$_{\rm dp}$ ($Y$-axis) in terms of CPU-Time for tasks that successfully conduct datapath propagation.

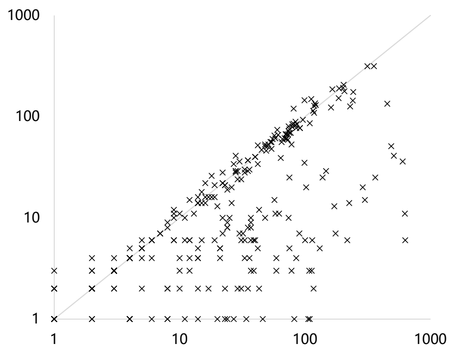

Fig. 11. AVR ($X$-axis) vs. AVR$_{\rm dp}$ ($Y$-axis) about the number of refinements for tasks that successfully conduct datapath propagation.

propagation. These lemmas consider the original semantics of datapath operations and tighten the datapath abstraction by adding constraints over UFs in the abstract state space.

Fig. 10 and Fig. 11 show the comparison results of AVR and AVR$_{\rm dp}$ in terms of CPU-time and the number of refinement on these 372 cases, respectively. In most cases, AVR$_{\rm dp}$ reduces the number of refinements by employing datapath propagation and achieves higher efficiency. Columns 7-14 in Tab. IV also report the statistics of these 372 cases, AVR spends 25186.2 seconds to verify these cases, and that number of AVR$_{\rm dp}$ is 8796.9 seconds – AVR$_{\rm dp}$ achieves 2.86x speedup than AVR. Moreover, AVR has 15651 refinements whereas AVR$_{\rm dp}$ has 10944 – applying datapath propagation reduces 30.1% refinements on these cases.

There are 317 both-verified cases that AVR$_{\rm dp}$ does not generate a datapath propagation lemma. However, these 317 cases only involve 276 refinements. After analysis, we found that the initial abstraction is accurate enough for these cases to prove correctness or find violations. So only a few refinements are called, and there is no room for AVR$_{\rm dp}$ to generate DPL during the propagation. These cases correspond to the points in Fig. 9 on the diagonal or slightly above the diagonal.

Tab. IV shows the comparative results of AVR and AVR$_{\rm dp}$ on cases that successfully conduct datapath propagation. The column `Config` lists the configurations of AVR and AVR$_{\rm dp}$. Note that *all* means both tools run all configurations in parallel. The tool reports the result and exits if any configuration can solve the verification task. Columns 3-7 display the data about the verified cases, where `Num` is the number of verified cases, followed by `Safe` and `Unsafe` that are the number of cases that satisfy or violate safety properties. `DPLs` is the number of generated datapath propagation lemmas. The last columns display statistics of tasks that both AVR and AVR$_{\rm dp}$ can verify. Column `Speedup` shows the speedup of AVR$_{\rm dp}$ compared to AVR. The speedup greater than 1.0x means that AVR$_{\rm dp}$ is faster than the selected tool. Columns `Refs` and `Percentage` are the number of refinements and the percentage of AVR$_{\rm dp}$ and AVR. The percentage lower than 100% means that AVR$_{\rm dp}$ reduces the number of refinements.

According to the first row, AVR$_{\rm dp}$ verifies 471 cases that successfully conduct datapath propagation. Among these cases, AVR can verify 372 of them. AVR$_{\rm dp}$ generates 7018 datapath propagation lemmas in total. The arithmetic-related DPLs include SUB$(\hat{0},\hat{1})$ != $\hat{0}$, ADD$(\hat{3},\hat{0})$!=$\hat{3}$, MINUS$(\hat{x}[53:36])$==$\hat{0}$ etc. The relational-related DPLs include LT$(\hat{0},\hat{1})$, $\neg$LE$(\hat{2},\hat{0})$, and LT$(\hat{0},\hat{x}) \vee \hat{x} == \hat{0}$, etc. DPLs related to Bitwise and Reduction operations include $\neg$ReductionOr$(\hat{0})$, ReductionAnd$(\hat{7})$, $\hat{x} == \hat{0} \vee \neg$ReductionOr$(\hat{x}) == \hat{0}$, etc. Meanwhile, some nested lemmas, e.g., SUB(ADD$(\hat{x},\hat{y})$, $\hat{x}$)=$\hat{y}$ can also be generated.

The following eight rows in Tab. IV display the comparative results of AVR and AVR$_{\rm dp}$ under different verification configurations. For CPU-time, the speedup is between 1.06x and 2.17x. Moreover, AVR$_{\rm dp}$ reduces the number of refinements between 17.15% to 41.31%. AVR$_{\rm dp}$ outperforms AVR in each configuration. It is obvious that, whether for AVR or AVR$_{\rm dp}$,

TABLE IV
THE EXPERIMENTAL RESULTS OF CASES WITH SUCCESSFUL DATAPATH PROPAGATION.

| Tool | Config | Verified | | | | Both-verified | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Num | Safe | Unsafe | DPLs | Num | Safe | Unsafe | CPU-Time (s) | Speedup | Refs | Percentage | DPLs |
| AVR | all | 372 | 366 | 6 | - | 372 | 366 | 6 | 25186.2 | - | 15651 | - | - |
| AVR$_{\text{dp}}$ | all | 471 | 464 | 7 | 7018 | 372 | 366 | 6 | 8796.9 | **2.86x** | 10944 | **69.93%** | 3923 |
| AVR | default | 250 | 246 | 4 | - | 250 | 246 | 4 | 11044.4 | - | 20883 | - | - |
| AVR$_{\text{dp}}$ | default | 312 | 308 | 4 | 2743 | 250 | 246 | 4 | 8525.8 | **1.30x** | 16580 | **79.39%** | 2206 |
| AVR | sa | 331 | 326 | 5 | - | 331 | 326 | 5 | 40304.3 | - | 1547 | - | - |
| AVR$_{\text{dp}}$ | sa | 388 | 382 | 6 | 5563 | 331 | 326 | 5 | 18556.5 | **2.17x** | 908 | **58.69%** | 3112 |
| AVR | split | 184 | 180 | 4 | - | 184 | 180 | 4 | 10135.0 | - | 16704 | - | - |
| AVR$_{\text{dp}}$ | split | 361 | 357 | 4 | 7246 | 184 | 180 | 4 | 7299.7 | **1.39x** | 12050 | **72.14%** | 2500 |
| AVR | sa8 | 283 | 279 | 4 | - | 283 | 279 | 4 | 17496.1 | - | 15296 | - | - |
| AVR$_{\text{dp}}$ | sa8 | 344 | 340 | 4 | 3002 | 283 | 279 | 4 | 16433.0 | **1.06x** | 11011 | **71.99%** | 2496 |
| AVR | sa16 | 290 | 286 | 4 | - | 290 | 286 | 4 | 27813.7 | - | 14761 | - | - |
| AVR$_{\text{dp}}$ | sa16 | 350 | 346 | 4 | 2968 | 290 | 286 | 4 | 20388.1 | **1.36x** | 10006 | **67.79%** | 2593 |
| AVR | sa32 | 322 | 317 | 5 | - | 322 | 317 | 5 | 33277.4 | - | 5137 | - | - |
| AVR$_{\text{dp}}$ | sa32 | 378 | 373 | 5 | 5200 | 322 | 317 | 5 | 27087.9 | **1.23x** | 4046 | **78.76%** | 2791 |
| AVR | level0 | 224 | 220 | 4 | - | 224 | 220 | 4 | 15296.8 | - | 19829 | - | - |
| AVR$_{\text{dp}}$ | level0 | 309 | 305 | 4 | 2415 | 224 | 220 | 4 | 10446.0 | **1.46x** | 13341 | **67.28%** | 1964 |
| AVR | level5 | 270 | 265 | 5 | - | 270 | 265 | 5 | 31738.8 | - | 48692 | - | - |
| AVR$_{\text{dp}}$ | level5 | 329 | 324 | 5 | 2755 | 270 | 265 | 5 | 25371.4 | **1.25x** | 40339 | **82.85%** | 2227 |

running multiple configurations in parallel and selecting the fastest one is more effective than running a single configuration. Each configuration may be more suitable for specific problems, and the combined running of multiple configurations can fully leverage the tool's performance, achieving better results in practice. Compared to AVR, AVR$_{\text{dp}}$ shows the greatest performance improvement under the configuration *all*.

The `sa` mode disables UF solving and utilizes the BV solver even in abstract mode. Preceding solver invocation, datapath propagation treats operations in $\varphi$ as uninterpreted functions and applies propagation rules. If unsat is yielded, further BV solver invocation for bit-blasting and abstract cube generation is unnecessary. The datapath propagation lemmas are stored in $\phi_{dpl}$ for subsequent verification. While these lemmas are BV tautologies, they may contain terms not in $\varphi$ but recorded during prior query propagation in $\mathcal{P}$. These terms introduce new elements for $\varphi$ solution, impacting SA+IC3's verification and refinement direction. AVR$_{\text{dp}}$ and AVR differ under `sa` config. For AVR, terms are introduced via refinement, potentially prematurely during datapath propagation, leading to fewer spurious counterexamples and refinement iterations.

## D. Discussions

***Scalability:*** The datapath propagation is orthogonal to the CEGAR framework and attempts to generate datapath propagation lemmas over datapath operations. Although we focus on the IC3 algorithm within the datapath abstraction and refinement framework, the DP-IC3 algorithm serves as an encoder that calls abstract formula queries on demand. Instead of taking all the datapath operations as uninterpreted and putting the heavy burden on CEGAR, lightweight strategies, or heuristics may convey essential information from a different perspective, guide the abstraction-based verification, and improve its scalability and efficiency.

***Threats to Validity:*** The main threats to our method's validity are whether the performance improvements are due to our tactic and whether our implementation and experimental results are credible. Firstly, we implement the proposed method in AVR and make a comparison with it. The improvements over AVR must come from our tactic. Secondly, the reduction in the number of refinements is consistent with the theoretical analysis, which confirms that the improvements are indeed from our approach. Thirdly, our implementation is loosely coupled with the original verification framework. Benchmarks are collected from the latest two HWMCCs, one of the most representative and convincing open sources in hardware verification. Moreover, we compare our method with the newest version of the state-of-the-art tools. We are thus confident in the effectiveness of our tactic.

***Limitations:*** We currently focus on datapath operations about arithmetic, relational, bitwise, shifting, and logical operations. We also develop some strategies to make it a lightweight and fast procedure. Therefore, our approach is sound but incomplete; it may not generate datapath propagation lemmas in some situations. To improve the scalability of the proposed method, we plan to elaborate on more propagations about arrays, concatenation, and extraction operations. Moreover, elaborated strategies and heuristics for generalizing datapath propagation lemmas are required to improve the overall efficiency further.

## VII. RELATED WORKS

### A. Advanced IC3-based approach

IC3 has been the most successful and widely applied technique for hardware verification in recent years. Various optimizations have been developed to improve the bit-level IC3 engine. PDR [2] proposes a simplified and faster implementation of IC3 by using three-valued simulation. PDR learns short clauses without numerous generalizations and achieves a significant speedup. UFAR [28] is a hybrid word- and bit-level solver that replaces heavy bit-level arithmetic logic with UF in Bounded model checking (BMC) [29] or in PDR.

However, the bit-level IC3 and its variants still suffer from the state space explosion problem. Many advanced abstraction techniques are proposed to lift the IC3 from bit-level to word-level. IC3IA [18] proposed a tight integration of IC3 with implicit abstraction, a form of predicate abstraction [30], [31]. With this technique, IC3 operates at the Boolean level of the abstract state space and generates inductive clauses over the abstraction predicates. It eliminates spurious counterexamples by incrementally generating and adding a set of new predicates. However, IC3IA may learn numerous predicates during the search to eliminate spurious counterexamples.

Averroes [6] integrates the IC3 with datapath abstraction, which can be seen as two layers of the CEGAR loop. The inner loop conducts IC3 on the abstract state space. The outer loop tightens the current abstraction by generating datapath refinement lemmas. These datapath lemmas refute the spurious counterexample that the inner loop returns. However, roughly abstracting all the datapath operations as UFs makes the verification framework lose their semantics, which may be useful for pruning the abstract state space. Chen [32] applies the knowledge of the control-flow graph in SMT solving, and Zpre [33] utilizes the knowledge of thread-interleaving to accelerate the concurrent program verification. Inspired by these works and considering the knowledge of datapath operations, we propose datapath propagation.

AVR [8], [14] extends Averroes with syntax-guided abstraction (SA), which encodes the abstract state space using the partition of UFs. IC3 with SA+UF allows for efficient reasoning regardless of the bit-width of variables or the datapath operations. AVR is the champion tool in the recent HWMCC. Currently, our method lacks support for datapath propagation involving bit-field extraction and concatenation. Instead, we adhere to the approaches employed in AVR, which incorporates a straightforward procedure introducing partial interpretation of extract/concat operations rather than abstracting them entirely as uninterpreted in the EUF logic. The consideration of accommodating these intricate yet commonly encountered operations is part of our future development plans. SyGuS-APDR [19], [34] utilizes syntax-guided synthesis to generate word-level lemmas heuristically. It includes a predefined grammar template and term production rules for generating candidate lemmas. These validated lemmas may prune the bad state space and tighten previous frontiers.

Reberto surveyed previous works [35] that focus on adding cheap but incomplete lemmas to the formula before invoking a complete but expensive decision procedure for a given theory $\mathcal{T}$. The static learning [36] technique suggests a prior some small and "obvious" $\mathcal{T}$-valid lemmas and drives the search direction of DPLL. Layered theory solver [37] try to establish the unsatisfiability of the current assignment $u$ in less expensive but much easier sub-theories. If a higher-level solver finds a conflict, this conflict is used to prune the search at the Boolean level; if it does not, the lower-level solver is activated to refine the former layer. The datapath propagation is a kind of lazy learning technique. It propagates the outcomes of datapath operations to the abstract state space by generating UF lemmas. These lemmas are cheap and incomplete, but they can guide the verification procedure to prune the bad state space and tighten the frontiers.

### B. *Constant propagation in Verification*

Constant propagation is an optimization technique commonly used in compilers and software analysis. The main goal is to replace variables or expressions with their constant values wherever possible, reducing the complexity of the system representation. Regarding formal verification, constant propagation can simplify the verification process and reduce the state space, leading to more efficient verification.

Armoni [38] uses constant propagation to simplify SAT formulas. They build expression graph (EG), a directed acyclic graph, for each BMC formula. The propagation starts from the leaves that denote variables or constants and updates the EG dynamically. Since BMC instants involve variables in different time frames, constant propagation is useful for pruning variables and reduces the complexity of the formulas. Wegman [17] proposes elaborated algorithms in flow analysis. Constants within conditional statements can be propagated if the conditions guarantee that a certain variable will always have a constant value under these conditions. This can lead to the elimination of branches and further state space reduction.

Different from the above applications, we focus on datapath operations in Verilog RTL design and perform constant propagation across concrete and abstract state space. In this way, we consider the original semantics of datapath operations, attain their outcomes, and propagate these results to corresponding UFs iteratively in abstract state space.

## VIII. Conclusion

This paper presents a datapath propagation mechanism for datapath abstraction-based hardware verification. We leverage concrete constant values to iteratively compute the outcomes of relevant datapath operations and their associated uninterpreted functions in the abstract state space. Meanwhile, we generate datapath propagation lemmas in abstract state space and tighten the datapath abstraction. We implemented the proposed method in a prototype tool named $\text{AVR}_{\text{dp}}$ and conducted experiments to compare $\text{AVR}_{\text{dp}}$ with state-of-the-art hardware verification tools. We collected benchmarks from hardware model checking competition 2019-2020. The experimental results show that our approach is effective and efficient.

## References

[1] A. R. Bradley, "Sat-based model checking without unrolling," in *VMCAI*, R. Jhala and D. Schmidt, Eds., Berlin, Heidelberg, 2011, pp. 70–87.

[2] N. Een, A. Mishchenko, and R. Brayton, "Efficient implementation of property directed reachability," in *2011 Formal Methods in Computer-Aided Design (FMCAD)*, 2011, pp. 125–134.

[3] R. Brayton and A. Mishchenko, "Abc: An academic industrial-strength verification tool," in *Computer Aided Verification*, T. Touili, B. Cook, and P. Jackson, Eds., Berlin, Heidelberg, 2010, pp. 24–40.

[4] Y. Vizel, O. Grumberg, and S. Shoham, "Lazy abstraction and sat-based reachability in hardware model checking," in *2012 Formal Methods in Computer-Aided Design (FMCAD)*, 2012, pp. 173–181.

[5] Y. Vizel and A. Gurfinkel, "Interpolating property directed reachability," in *International Conference on Computer Aided Verification*. Springer, 2014, pp. 260–276.
[6] S. Lee and K. A. Sakallah, "Unbounded scalable verification based on approximate property-directed reachability and datapath abstraction," in *Computer Aided Verification*, Cham, 2014, pp. 849–865.
[7] A. Goel and K. Sakallah, "Model checking of verilog rtl using ic3 with syntax-guided abstraction," in *NASA Formal Methods*, J. M. Badger and K. Y. Rozier, Eds., Cham, 2019, pp. 166–185.
[8] A. Goel, "From finite to infinite: Scalable automatic verification of hardware designs and distributed protocols," Ph.D. dissertation, 2021.
[9] R. Hojati and R. K. Brayton, "Automatic datapath abstraction in hardware systems," in *Computer Aided Verification*, P. Wolper, Ed., Berlin, Heidelberg, 1995, pp. 98–113.
[10] C. Barrett, P. Fontaine, and C. Tinelli, "The Satisfiability Modulo Theories Library (SMT-LIB)," www.SMT-LIB.org, 2016.
[11] C. W. Barrett, D. L. Dill, and J. R. Levitt, "A decision procedure for bit-vector arithmetic," in *Proceedings of the 35th Annual Design Automation Conference*, 1998, pp. 522–527.
[12] V. Ganesh and D. L. Dill, "A decision procedure for bit-vectors and arrays," in *Computer Aided Verification: 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007. Proceedings 19*. Springer, 2007, pp. 519–531.
[13] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *Computer Aided Verification*, E. A. Emerson and A. P. Sistla, Eds., Berlin, Heidelberg, 2000, pp. 154–169.
[14] A. Goel and K. Sakallah, "Avr: Abstractly verifying reachability," *Tools and Algorithms for the Construction and Analysis of Systems*, vol. 12078, p. 413, 2020.
[15] H. Ma, A. Goel, J.-B. Jeannin, M. Kapritsos, B. Kasikci, and K. A. Sakallah, "I4: Incremental inference of inductive invariants for verification of distributed protocols," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, ser. SOSP '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 370–384. [Online]. Available: https://doi.org/10.1145/3341301.3359651
[16] D. Callahan, K. D. Cooper, K. Kennedy, and L. Torczon, "Interprocedural constant propagation," *ACM SIGPLAN Notices*, vol. 21, no. 7, pp. 152–161, 1986.
[17] M. N. Wegman and F. K. Zadeck, "Constant propagation with conditional branches," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 13, no. 2, pp. 181–210, 1991.
[18] A. Cimatti, A. Griggio, S. Mover, and S. Tonetta, "Ic3 modulo theories via implicit predicate abstraction," in *Tools and Algorithms for the Construction and Analysis of Systems*, E. Ábrahám and K. Havelund, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 46–61.
[19] M. Mann, A. Irfan, F. Lonsing, Y. Yang, H. Zhang, K. Brown, A. Gupta, and C. Barrett, "Pono: A flexible and extensible smt-based model checker," in *Computer Aided Verification*, A. Silva and K. R. M. Leino, Eds. Cham: Springer International Publishing, 2021, pp. 461–474.
[20] Y. Xia, A. Becchi, A. Cimatti, A. Griggio, J. Li, and G. Pu, "Searching for i-good lemmas to accelerate safety model checking," in *Computer Aided Verification*, C. Enea and A. Lal, Eds. Cham: Springer Nature Switzerland, 2023, pp. 288–308.
[21] L. De Moura and N. Bjørner, "Satisfiability modulo theories: Introduction and applications," *Commun. ACM*, vol. 54, no. 9, p. 69–77, sep 2011. [Online]. Available: https://doi.org/10.1145/1995376.1995394
[22] P. Bjesse and K. Claessen, "Sat-based verification without state space traversal," in *Formal Methods in Computer-Aided Design*, W. A. Hunt and S. D. Johnson, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 409–426.
[23] M. Sheeran, S. Singh, and G. Stålmarck, "Checking safety properties using induction and a sat-solver," in *Formal Methods in Computer-Aided Design*, W. A. Hunt and S. D. Johnson, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 127–144.
[24] K. L. McMillan, "Interpolation and sat-based model checking," in *Computer Aided Verification*, W. A. Hunt and F. Somenzi, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 1–13.
[25] A. Niemetz, M. Preiner, C. Wolf, and A. Biere, "Btor2 , btormc and boolector 3.0," in *Computer Aided Verification*, H. Chockler and G. Weissenbacher, Eds., Cham, 2018, pp. 587–595.
[26] C. Wolf, "Yosys open synthesis suite," https://yosyshq.net/yosys/.
[27] A. Biere, K. Heljanko, and S. Wieringa, "AIGER 1.9 and beyond," Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstr. 69, 4040 Linz, Austria, Tech. Rep. 11/2, 2011.
[28] Y.-S. Ho, P. Chauhan, P. Roy, A. Mishchenko, and R. Brayton, "Efficient uninterpreted function abstraction and refinement for word-level model checking," in *FMCAD*, 2016, pp. 65–72.
[29] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, "Bounded model checking." *Handbook of satisfiability*, vol. 185, no. 99, pp. 457–481, 2009.
[30] T. Ball, A. Podelski, and S. K. Rajamani, "Boolean and cartesian abstraction for model checking c programs," in *Tools and Algorithms for the Construction and Analysis of Systems*, T. Margaria and W. Yi, Eds. Springer Berlin Heidelberg, 2001, pp. 268–283.
[31] S. Graf and H. Saidi, "Construction of abstract state graphs with pvs," in *Computer Aided Verification*, O. Grumberg, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 72–83.
[32] J. Chen and F. He, "Control flow-guided smt solving for program verification," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 351–361. [Online]. Available: https://doi.org/10.1145/3238147.3238218
[33] H. Fan, W. Liu, and F. He, "Interference relation-guided smt solving for multi-threaded program verification," in *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 163–176. [Online]. Available: https://doi.org/10.1145/3503221.3508424
[34] H. Zhang, A. Gupta, and S. Malik, "Syntax-guided synthesis for lemma generation in hardware model checking," in *Verification, Model Checking, and Abstract Interpretation: 22nd International Conference, VMCAI 2021, Copenhagen, Denmark, January 17–19, 2021, Proceedings*. Berlin, Heidelberg: Springer-Verlag, 2021, p. 325–349. [Online]. Available: https://doi.org/10.1007/978-3-030-67067-2_15
[35] R. Sebastiani, "Lazy satisfiability modulo theories," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 3, no. 3-4, pp. 141–224, 2007.
[36] A. Armando, C. Castellini, and E. Giunchiglia, "Sat-based procedures for temporal reasoning," in *European Conference on Planning*. Springer, 1999, pp. 97–108.
[37] R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, Z. Hanna, A. Nadel, A. Palti, and R. Sebastiani, "A lazy and layered smt () solver for hard industrial verification problems," in *International Conference on Computer Aided Verification*. Springer, 2007, pp. 547–560.
[38] R. Armoni, L. Fix, R. Fraer, T. Heyman, M. Vardi, Y. Vizel, and Y. Zbar, "Deeper bound in bmc by combining constant propagation and abstraction," in *2007 Asia and South Pacific Design Automation Conference*, 2007, pp. 304–309.

**Hongyu Fan** is a PhD student at the School of Software of Tsinghua University. His received a B.E. degree from College of Computer Science and Technology, Jilin University in 2018. His research interests include concurrent program verification, hardware model checking, and SAT/SMT solving. He has published several papers in academic journals and international conferences, including PLDI, PPoPP, TOPLAS.

**Fei He** received the B.S. degree in computer science and technology from National University of Defense Technology in 2002, and the PhD. degree in computer science and technology from Tsinghua University in 2008. He is currently an Associate Professor in the school of software at Tsinghua University, Beijing, China. His research interests include model checking, program verification and automated logic reasoning.